Direct Neighbor Search

Jilian Zhang, Kyriakos Mouratidis, HweeHwa Pang

School of Information Systems, Singapore Management University, 80 Stamford Road, Singapore 178902
{jilian.z.2007, kyriakos, hhpang}@smu.edu.sg

Abstract

In this paper we study a novel query type, called *direct neighbor* query. Two objects in a dataset are direct neighbors (DNs) if a window selection may exclusively retrieve these two objects. Given a source object, a DN search computes all of its direct neighbors in the dataset. The DNs define a new type of affinity that differs from existing formulations (e.g., nearest neighbors, nearest surrounders, reverse nearest neighbors, etc) and finds application in domains where user interests are expressed in the form of windows, i.e., multi-attribute range selections. Drawing on key properties of the DN relationship, we develop an I/O optimal processing algorithm for data indexed with a spatial access method. In addition to plain DN search, we also study its K-DN and all-DN variants. The former relaxes the DN condition – two objects are K-DNs if a window query may retrieve them and only up to $K-1$ other objects – whereas the all-DN variant computes the DNs of every object in the dataset. Using real, large-scale data, we demonstrate the efficiency and practicality of our approach, and show that it vastly outperforms a competitor constructed from previous work.

Keywords: Direct Neighbors, Window Query, Low-dimensional Search

1. Introduction

In this work we focus on systems and applications where users browse databases via window queries. Consider a database where objects correspond to available services or products and are represented as rectangles in a d-dimensional space. A window query retrieves all objects that fall inside a user-specified axis-parallel rectangle. Fig. 1(a) illustrates a database with 10 objects in two-dimensional space. W_1, shown with a dashed border, is an example of a window query that returns r_7 and r_{10} in the result.

Alternative query types, such as nearest neighbors (NN) [1] and reverse nearest neighbors (RNN) [2], browse data based on the notion of spatial distance, provided that objects bear geographic coordinates. Inherent in the distance notion is the assumption that different dimensions can be combined in a predetermined way into Euclidean distance or another L^p metric.

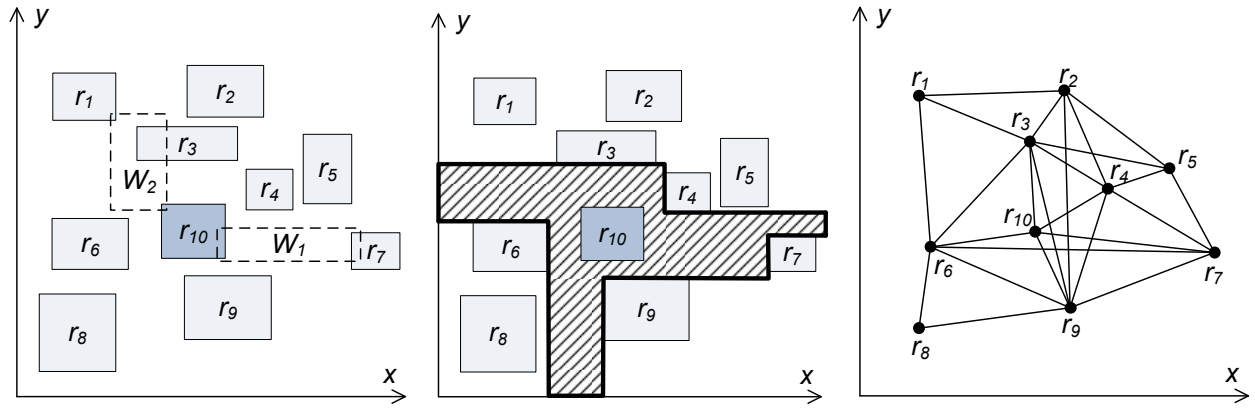
However, in many settings the data dimensions represent different aspects of the problem and are not directly comparable to each other. Thus, it is not meaningful to combine dimensions into a distance measure in determining the similarity between objects. In such settings, window queries are the only reasonable representation of user interests. An example is on-line property agencies like propertyguru.com.sg, on which owners, agents and developers post details of units for rental/sale. Potential buyers/tenants may browse available options by specifying ranges of their desired price and floor-area requirements (i.e., via window queries). For instance, in Fig. 1(a) the two dimensions could correspond to the rent and floor area, respectively. Another example is kayak.com. In this portal, users planning to fly between two cities may browse the available flight options by specifying acceptable ranges for the price and duration of the flight.

Since user interests are captured by window queries, similarity ought to be defined based on windows and, specifically, *on the potential of data objects to co-exist in the same query result*. Assume that the objects in Fig. 1(a) correspond to alternative services/products. To identify the immediate alternatives to r_{10} (called the *source*), its provider/manufacturer would want to know which objects are likely to be retrieved together with r_{10} by user queries. Consider alternatives r_3 and r_1. On one hand, there exist windows that would retrieve only r_3 and the source (r_{10}). On the other hand, for a query to report the source and object r_1, it must necessarily report r_3 as well. In this aspect, r_3 is a more immediate competitor/alternative to r_{10} than r_1. To capture this fact, we define direct neighbors as follows.

Definition 1. *Given a dataset S, we define as* ***direct neighbor*** *(DN) of a source object q any other data object $r \in S$ which may be exclusively retrieved (along with q) by a window query. In other words, there exists an axis-parallel window that overlaps only with q and r.*

A DN query at source q retrieves all its DNs in S. In the example of Fig. 1(a), for source object r_{10}, the result comprises r_3, r_4, r_6, r_7 and r_9. Applications of the DN query include competitor and marketability analysis, recommendation of alternatives, etc.

Competitor Analysis: Identifying the DNs of q could be used to improve its competitiveness with respect to directly comparable products/services [3], e.g., via competitor-aware advertisement or appropriate reconfiguration/redesign of q itself. For



(a) DN example (b) Exclusive retrieval region (c) All-DN example

Figure 1: DN and all-DN search

instance, the marketing team behind a property (or the airline offering a flight) would be interested in knowing which its immediate competitors are with respect to the rent-area criteria (airfare-duration), and potentially reconsider its pricing.

In certain dimensions there may be a clear preference direction (i.e., higher/larger values may be more desirable). For example, in the property scenario one could assert that lower price (equivalently, larger size) is generally preferable. The DN query, being independent of preference directions (if any), would also report properties that are costlier and smaller than the source q, i.e., theoretically less preferable. Such DNs are also useful for competitor analysis because they may indicate a potential to, say, mark up the price of q, or more aggressively advertise it against these competitors, or take into account (qualitative) factors other than price/area that may be involved in a client's decision. While clear preference directions may or may not exist in the data dimensions[1], this is irrelevant to DN retrieval, its semantics and its applicability.

Exclusive Retrieval Region: The DNs of a source object q also demarcate its *exclusive retrieval region*. Any window query that completely lies in this region and overlaps q is guaranteed to only overlap q. That is, the exclusive retrieval region defines the maximal search area where q is the only result of a window query, and by itself provides an indication of the competitiveness and marketability of q. Fig. 1(b) shows the exclusive retrieval region of source r_{10}. The region is delineated by the DNs of r_{10} (i.e., r_3, r_4, r_6, r_7 and r_9). Its derivation is discussed later in the paper.

Recommendation of Alternatives: In a system where user queries are expressed by windows, the DNs are natural candidates for alternative recommendations. That is, if a user is currently viewing object q, the search portal could suggest the DNs of q as alternatives for consideration. Alternative recommendations are common in property, flight or hotel room search systems, such as tripadvisor.com and booking.com (where users may browse accommodation options based on price and average user ratings).

The rationale behind DN formulation is that (since user interests are captured by windows) the similarity or comparability between two objects q and r is determined by the number of intervening objects retrieved by any window query overlapping q and r. In this regard, the definition can be generalized to provide a partial ordering of competitors based on the number of intervening objects. That is, the fewer the intervening objects, the more immediate threat posed by a competitor. This motivates the K-DN formulation, which reaches a broader set of alternatives by relaxing the DN condition.

Specifically, an object $r \in S$ is a K-DN of source q if there is a query window that intersects r, q and fewer than K other objects in S. K may or may not be known in advance. The latter case entails *incremental* K-DN processing where K can be incremented iteratively without the need to run the query from scratch, instead resuming it from where it last stopped. K-DN search finds application in scenarios similar to plain DN, the difference being that the scope of, say, competitor analysis is wider so that more alternatives are taken into consideration.

Another variant of DN search with practical relevance is the all-DN query. An all-DN query computes the DN set of every object in dataset S. As we explain later, the DN relationship is symmetric. Therefore, the output of the all-DN query may be visualized as an undirected graph in which the nodes correspond to objects, and the edges to instances of DN relationship. Fig. 1(c) illustrates the all-DN result for our 10-rectangle example. In addition to applications of the plain DN query, all-DN search can aid in optimizing a spatial index for window query processing, as follows.

Suppose that a data-partitioning index (e.g., an R-tree [5, 6]) is to be bulk-loaded with the 10 rectangles depicted in Fig. 1(a), i.e., to be built bottom-up from scratch [7, 8]. As follows directly from the definition of DN relationship, objects that are direct neighbors of each other are likely to be accessed by the same window query. Hence, it is beneficial (in lowering I/O cost) to store them in the same leaf node of the index. This

[1]For instance, a dimension could be the storey number where preference of lower, higher or middle floors is a personal choice [4].

can be achieved by computing the DN graph in Fig. 1(c), then performing a min-cut clustering (while constraining the cluster sizes by the leaf node capacity) before assigning object clusters to the leaves. Additional bulk-loading heuristics could also be formulated on the basis of the DN graph. A straightforward, yet inefficient way to answer the all-DN query is to perform a plain DN search for each $r \in S$. We develop an algorithm that improves performance by three orders of magnitude compared to this naïve solution.

The DN definition is irrelevant to the shape of data objects. We center on rectangular objects with axis-parallel sides, although our techniques also apply to point data and arbitrarily shaped objects (see Sec. 3.4). Our focus on rectangles is because they are more general than points and probably the most common/intuitive representation of services and products in multi-dimensional spaces. The extent of a data object in a dimension could capture a degree of fuzziness or its intrinsic association with a range of values. For example, each flight in kayak.com is associated with the range of prices that are offered by different online ticketing agencies (and potentially in different ticket classes). Hotel options in tripadvisor.com are associated with a range for price per night, depending on the exact dates of visit, etc.

As we show later in this paper, the DN problem and its variants are meaningful in low-dimensional spaces. We therefore assume that dataset S is indexed by a spatial access method, such as the R-tree. Our contributions are summarized as follows:

- We introduce and formalize a new query (DN) and its variants (K-DN and all-DN);
- We devise I/O optimal algorithms for DN and K-DN queries over data organized by a spatial index;
- We develop a sophisticated algorithm for all-DN processing that outperforms by orders of magnitude a repetitive application of DN search.

The rest of the paper is organized as follows. Sec. 2 surveys related work. Sec. 3 introduces DN processing, while Sec. 4 and 5 present our K-DN and all-DN algorithms, respectively. Sec. 6 then extends DN search to higher dimensions. Sec. 7 empirically evaluates our techniques, and Sec. 8 concludes the paper.

2. Related Work

DN processing has not been studied before, yet here we review related query types, such as nearest neighbor, nearest surrounder, and skyline queries. We also survey the segment tree, a data structure that we adapt for our framework.

A *nearest neighbor* query (NN) retrieves from a spatial dataset the object that lies closest to a user-specified source point q. For NN processing over datasets indexed by a spatial access method, the *depth-first* and *best-first* paradigms have been considered in [1] and [9], respectively. The latter is shown to be superior in I/O cost. Assume that the dataset is indexed by an R-tree. Starting from the root of the R-tree, encountered index entries e are pushed into a min-heap with $mindist(e, q)$ as key, i.e., the minimum distance between the source point q and the minimum bounding rectangle (MBR) of e. Iteratively, the top entry of the heap is popped and its corresponding R-tree node is accessed from disk; then, its child entries are en-heaped. The process is repeated until the first data entry (object) is popped and reported as the NN. If more nearest neighbors are required, the process continues and the next data entry popped is the second NN, and so on. The method is *incremental* in that it can keep reporting the next NN without needing to specify in advance how many neighbors are required in total. Also, it is I/O optimal, i.e., it fetches from disk the minimum possible number of R-tree nodes.

Another related problem is *nearest surrounder* search (NS) [10]. Given a spatial dataset S and a source point q, an NS query retrieves a set $\mathcal{R}_{\mathcal{NS}} \subseteq S$ of objects, each being the nearest neighbor of q with the scope of interest constrained at some range of angles around q. Objects in $\mathcal{R}_{\mathcal{NS}}$ collectively cover the whole angle range $[0°, 360°]$ around q; these objects have a clear line of sight from q, unblocked by other objects. The NS query differs from the conventional NN query in taking into account directional information of the nearest neighbors. The method proposed in [10] uses an angular sweeping technique to process the R-tree that indexes S. This approach also extends to K-tier NS retrieval, where the line of sight between q and each NS object may cross up to $K - 1$ others. NS (and K-tier NS) methods exist only for two dimensions and for point (zero-extent) source objects. NS search and similar visibility queries (e.g., [11]) are different by definition from our problem, as we also elaborate in Sec. 3.1. In that section, however, we devise a baseline DN approach (which works only for two-dimensional data) that uses NS as a building block.

The DN query also relates to preference handling, which aims to formulate and capture, qualitatively or quantitatively, user preferences [12, 13, 14]. An example is to deduce that the user prefers option A to option B, if attributes of A are 'better' than those of B according to some predefined preference relationship on attribute values. A qualitative preference model typically employs binary relationships such as partial ordering between options. In contrast, a quantitative model captures preferences via scoring functions on option attributes [12]. Various tools (called operators) have been devised to retrieve the most preferable records from databases, including *winnow* [12], *best* [15], and preference selection [16]. These operators are based on built-in predicates of the SQL language or designed as standalone functions embedded into SQL systems [17]. DN search does not fall under either category of preference handling, since it does not rely on any partial order or preference function model.

Work on *skyline* processing and its variants ([18, 19, 20]) is also relevant to ours. Consider a dataset S where each object has two attributes, x and y. An object here could correspond to a transportation option between two specific cities, with attributes price (x) and total duration (y). Assume that all options have different x and y values. An object (travel option) r is said to *dominate* another object r' if both of r's attributes are no larger than those of r'. Essentially, this implies that option

r is preferable to r' because the former is both cheaper and faster. The *skyline* of S comprises all objects that are not dominated by any other object. *Branch-and-bound skyline* (BBS) is an I/O optimal skyline algorithm [20] that utilizes an R-tree on S. BBS accesses the tree nodes in ascending $mindist$ order from the most "preferable" corner of the data space. In our example, this corner is the origin of the data space. Once a data object is found, it is added to the skyline. Subsequently encountered R-tree nodes (or objects) are accessed (included in the skyline, respectively) only if they are not dominated by any object currently in the skyline. K-skyband is a generalization of the skyline that includes all objects dominated by fewer than K others. BBS extends to K-skyband computation, retaining its I/O optimality.

The skyline query can be used for recommendation of alternatives. However, its semantics (and therefore its domain of applicability) is different from DN. The skyline operator is not input-sensitive, meaning that the result is *always the same* and it *does not depend on any user input*. In our transportation options example, the skyline options (i.e., those not dominated by any other in the input dataset) are only dependent on the dataset itself. No input is required from the user nor is there any way to alter the skyline result. Instead, the DN query is input-sensitive – the DN result depends on the source object q and varies with its extent and location. Another key difference is that the DN query pertains (and offers an auxiliary decision support mechanism) to systems where the users browse options via window queries. In contrast, the skyline operator requires simply a fixed and monotonic preference order in each data dimension (e.g., the smaller the price/duration of a travel option the better). Despite the differences in semantics, nature, and application domain, our processing techniques utilize an (adapted) skyline algorithm as a building block to derive a subset of DNs.

The *dynamic skyline* receives as input, in addition to data, a set of query objects[2]. Each data object is represented by the vector of its distances from every query object. A data object belongs to the dynamic skyline if its distance vector is not dominated by that of any other data object. The distances between data and query objects can be Euclidean [21], road network distances [22] or general metric distances [23]. The problem differs from ours in that (i) DN search involves a single input dataset (data objects only), (ii) dynamic skylines are defined over (distance) vectors whereas the DN relationship is defined over rectangles and, most importantly, (iii) DN search captures the exclusive co-existence of two objects in the result of a window query instead of the dominance (or not) between them.

In a sense, DN search is related to influence set computation, i.e., identification of objects that could affect or be affected by a source object q. The concept of influence sets was introduced in [2], and formulated as a *reverse nearest neighbor* search (RNN) at q. The RNN set of q includes those objects r in a dataset S that have q as their nearest neighbor. RNN processing has received significant attention; [24] provides a comprehensive survey of existing work.

[2]The concept of dynamic skyline was introduced in [20] to refer to a more general problem. We focus on its spatial versions due to their higher relevance to DN formulation.

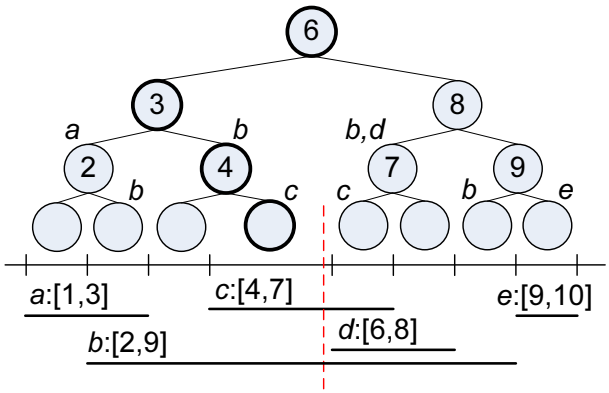


Figure 2: Example of segment tree and stabbing query at $q = 5.8$

By definition, the DN query retrieves different objects than RNN. There is an interesting similarity though – window queries in DN play the role of NN queries in RNN. Specifically, given a source object q, DN search (or RNN search) discovers those objects around which, if a window query (a NN query, respectively) is issued, the result will exclusively include q. This correspondence implies that the type of influence stemming from the DN relationship is meaningful in domains where user interests are expressed by window queries, whereas the influence derived from the RNN relationship is meaningful in applications where users browse data by NN queries.

There is a similar analogy with the *reverse top-K* query [25] too. In the reverse top-K problem, the input comprises a dataset S and a set of scoring functions. Each function is defined over the attributes of S and assigns a score to every object $r \in S$. The objective is to determine, for a source object $q \in S$, which of the functions include it in their top-K result. This problem cannot be mapped to ours. For example, the top-K queries/functions are known in advance, and the influence of q relates to functions rather than other objects in S. However, it is interesting that a reverse query notion (top-K in this case) is used to discover entities (functions) influenced by the source.

The *reverse skyline* query is defined in [26]. That work considers a spatial version of dominance before reversing it. Specifically, the input includes a source point q and a set of d-dimensional data points. A data point r is said to *dynamically dominate* another r' with respect to q if the projection of r on each of the d axes lies closer to q than the corresponding projection of r'. Now, a data point r belongs to the reverse skyline of q if q is not dynamically dominated by any other point with respect to r. Although again a connection exists with our problem, here we consider window queries instead of dynamic dominance. To further stress the difference, note that reverse skyline is not a symmetric relationship (i.e., the fact that r belongs to the reverse skyline of q does not mean that the converse holds too). In contrast, the DN relationship is symmetric, as explained in Sec. 5.

The definition of the all-DN graph resembles to some extent the concept of *Gabriel graph* in computational geometry

[27]. Given a set S of points in the Euclidean plane, the Gabriel graph G uses S as its vertex set. There is an edge between nodes (i.e., points) r and r' if and only if the *circle* whose diameter has r and r' as endpoints is empty. A fundamental difference between all-DN graph and Gabriel graph is that the latter is defined strictly for points, not rectangles. Another distinction is that its edges represent empty circles (with specific, fixed diameter) versus axis-parallel rectangular windows (with arbitrary diagonal). The different edge definition leads to different topologies; for instance, the Gabriel graph is planar (no edge intersects another) which is not the case in all-DN graph (e.g., see Fig. 1(c)). The Gabriel graph could be defined under the L^∞ norm (instead of the Euclidean), where an edge between points r and r' exists if and only if the *square* with diagonal corners r and r' is empty. The problem is still defined only for points (and the Gabriel graph remains planar). The edges correspond to empty squares with a specific diagonal versus completely arbitrary axis-parallel windows (with unknown diagonal and arbitrary side-length proportions).

Our algorithms rely on the *segment tree* [28]. This is a balanced binary tree used for efficiently answering stabbing queries on a set of line segments, i.e., reporting all segments in a one-dimensional space that envelop a given query point. Consider a set of line segments $S = \{s_1, s_2, ..., s_N\}$, each delimited by two endpoints. To construct a segment tree on S, we sort the $2N$ different endpoints into an ascending sequence $P = (p_1, p_2, ..., p_{2N})$. The endpoints in P divide the one-dimensional space $(-\infty, \infty)$ into $2N + 1$ *atomic intervals*. A binary tree T is constructed bottom-up, with the leftmost leaf node covering the leftmost interval $(-\infty, p_1]$, the second leaf covering $(p_1, p_2]$, and so on. Each internal node in T covers the union of the intervals of its two children. Every (internal or leaf) node n_j in T stores a *segment list* L_j containing those segments of S that completely cover the node's interval but not the interval of its parent. To answer a stabbing query at point q, T is traversed from the root, reporting all the segments in the lists of those nodes n_j whose interval envelops q. The segment tree has $O(N \log N)$ construction time, $O(\log N)$ insertion time, and $O(\kappa + \log N)$ query time where κ is the number of segments in the result.

Fig. 2 illustrates a segment tree built on five line segments, a, b, c, d, and e. The line segments and their exact intervals are depicted at the bottom of the figure. Inside each tree node we draw the corresponding splitting value (implicitly defining its covering interval), and next to the node we present its segment list. Suppose the user issues a stabbing query at point $q = 5.8$, shown as a vertical dashed line. The search begins from the root. Comparison with the splitting value therein (i.e., 6) directs the search to the left child, and in turn, from that node to its right child, and so on until it reaches a leaf node. The visited nodes are shown in bold border. The union of the segment lists in visited nodes forms the query result, i.e., segments $\{b, c\}$.

3. Direct Neighbor Search

Given a dataset S and a source object q in a d-dimensional space, a data object $r \in S$ is a *direct neighbor* (DN) of q if there exists a (d-dimensional) axis-parallel window that intersects only q and r. Our focus is on low-dimensional spaces – as we explain in Sec. 6.1, the DN problem is meaningful when dimensionality is low, because the number of DNs grows quickly with d. For ease of presentation, we consider two dimensions before extending our methodology to more (in Sec. 6). We assume rectangular (source and data) objects with axis-parallel sides. Notwithstanding this, our work applies to point data, which may be treated as zero-extent rectangles, and to arbitrarily shaped objects (discussed in Sec. 3.4). The objects may or may not overlap. We target disk-resident datasets S, organized by a spatial index like the R-tree.

Table 1: Notation

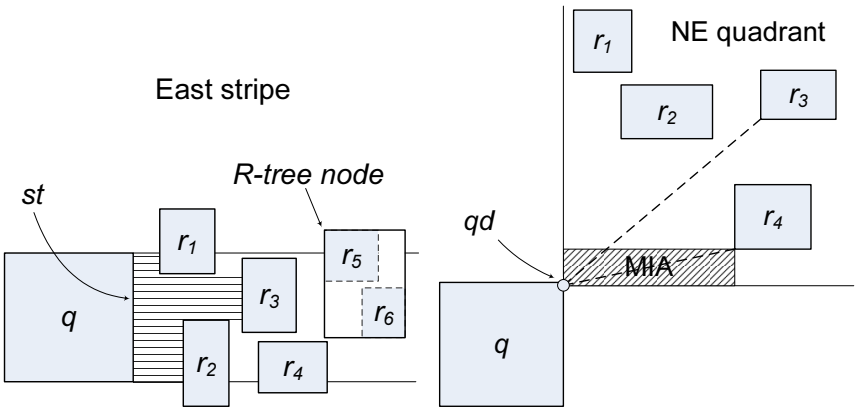
Symbol	Description
S	set of data objects
N	cardinality of S
T_x, T_y	segment trees on x- and y-extents of objects
L_i	the object list of leaf node n_i in a segment tree
I_i	interval covered by leaf node n_i in a segment tree
n_-, n_+	boundary leaf nodes in a segment tree

Suppose that the data space is $[0, X_{max}][0, Y_{max}]$ and the extent of the source q is $[q.x_l, q.x_h][q.y_l, q.y_h]$. We perform DN retrieval in the four *stripes* and four *quadrants* of q. The east stripe is the area defined by the right edge of q, extending horizontally to the right border of the data space, i.e., the area $[q.x_h, X_{max}][q.y_l, q.y_h]$. The east stripe of an example source q is shown in Fig. 3(a). The north-east quadrant (NE) is the axis-parallel area extending diagonally from the NE corner of q to the NE corner of the data space, i.e., $[q.x_h, X_{max}][q.y_h, Y_{max}]$. Fig. 3(b) shows the NE quadrant of an example source q. The other stripes and quadrants are defined accordingly. Note that q, the four stripes and the four quadrants define a partition of space into 9 regions. We first consider DN search inside the quadrants, proposing two methods for this, then in the stripes. Objects that intersect q are directly reported as DNs (we establish the convention that all of these objects belong to the DN set).

For simplicity, we initially assume that objects in S fall completely inside a single stripe or quadrant. This assumption is relaxed in Sec. 3.3. Our objective is to minimize the total processing cost, comprising I/O and CPU time. Table 1 lists the frequently used notation.

3.1. DN Search in Quadrants

In Fig. 3(b), point qd marks the NE corner of q. A preliminary approach to derive the DNs in the NE quadrant is to issue a *constrained* NS search at qd, limiting the visibility search to the 90° angle NE of qd. The surrounders derived are a superset of the DNs (in NE quadrant). To see this, if rectangle r is a DN of q, by definition there is a query window that exclusively intersects r and q. Since r is in the NE quadrant, this query window must include qd. Also, because this window intersects no other object, there is visibility between qd and r. Therefore, the quadrant DNs are also NSs. On the contrary, an NS is not



(a) Stripe DN (b) Quadrant DN

Figure 3: DN search in quadrants and stripes

always a DN; e.g., r_3 is visible from qd but not a DN (every query window overlapping r_3 and q necessarily intersects r_2 and r_4 too). Before presenting how false positives (NSs that are not DNs) can be eliminated, we define the notion of minimum intersection area.

Definition 2. *The **Minimum Intersection Area (MIA)** of an object r (lying in a specific quadrant) is the axis-parallel rectangle defined by qd and the closest corner of r to qd.*

The striped area in Fig. 3(b) is the MIA of r_4. It is easy to see that any query window that intersects an object r and q *must* envelop the MIA of r. This leads to the following crucial observation.

Observation 1. *An object r (lying in a specific quadrant) is a DN if and only if its MIA intersects no other object.*

Observation 1 helps to disqualify NSs that are not DNs. Let $\mathcal{R}_{\mathcal{NS}}$ be the set of constrained NSs. In a straightforward application of the observation, we may check for every $r \in \mathcal{R}_{\mathcal{NS}}$ whether its MIA intersects any other object in S; if so, r is disqualified. The remaining NSs are the DNs. This approach would require multiple window queries in S (as many as the total number of NSs), incurring considerable I/O overhead. On a closer inspection, the overhead can be eliminated. Specifically, if there exist objects in S that overlap the MIA of a candidate $r \in \mathcal{R}_{\mathcal{NS}}$, at least one of them will be visible from q, and therefore already in $\mathcal{R}_{\mathcal{NS}}$. This implies that we need to check the MIA of r only against objects in $\mathcal{R}_{\mathcal{NS}}$, rather than against the entire S.

We term the above approach *constrained NS* (CNS). Its main drawback is that it accesses a superset of the strictly needed objects (e.g., r_3 in Fig. 3(b)). In our experiments with real and synthetic data (Table 4) the number of NSs is an order of magnitude larger than that of actual quadrant DNs, leading to a large false positive ratio. Accessing these false positives translates to unnecessary I/Os. Also, NS queries require a significant amount of computations, due to the angular sweeping mentioned in Sec. 2. Observation 1 paves the way for a more efficient (and I/O optimal) method, named *skyline DN* (SDN), via Lemma 1.

Lemma 1. *The DNs of q in a quadrant are exactly the skyline objects in this quadrant, with qd as origin and each object represented by its closest corner to qd.*

Proof 1. *By definition, a point p is dominated by those and only those points p' that fall inside the rectangle defined by p and the origin, called dominating rectangle of p. Thus, p is a skyline point if and only if its dominating rectangle is empty. In our context, since the skyline is defined on the objects' corners that are closest to qd, the dominating rectangles of these corners correspond to the MIAs of the respective objects. From Observation 1 it follows that the skyline and quadrant DN sets are identical.*

Based on Lemma 1, SDN computes the DNs in each quadrant using the I/O optimal BBS algorithm [20].

3.2. DN Search in Stripes

An object r is a DN of q in a stripe, say the east, if a rectangle can be found to exclusively intersect r and q's right edge st, i.e., there exists a horizontal ray from some point on st to a point on r's left edge that intersects no other object. In Fig. 3(a), for example, the east stripe DNs of q are r_1, r_2, r_3. To identify such DNs, we sweep from st to the right, examining objects r (in the east stripe) in increasing $r.x_l$ order. For each object r encountered, we check if the previously considered objects in this stripe (collectively) block r from st. If not, r is a DN. The search terminates when the DNs discovered so far block all the remaining objects in the stripe.

To iteratively discover objects in increasing $r.x_l$ order, we utilize the R-tree on S. The process is similar to a best-first incremental NN search at q, as described in Sec. 2, where the sorting key of the min-heap is $mindist$ from st. The difference is that the search is constrained to the east stripe and that R-tree nodes that are blocked from st by already encountered DNs are

pruned (i.e., not visited). In Fig. 3(a), for instance, the illustrated R-tree node that holds r_5 and r_6 is pruned (not accessed at all) during the search because it is fully blocked by the previously reported DNs r_1, r_2, r_3. Object r_4, although popped from the search heap, also fails the visibility check and is excluded from the DN set. The I/O optimality of the best-first NN algorithm in [9], in conjunction with the pruning of blocked nodes, guarantees optimality for the stripe DN search too.

Lemma 2. *The stripe DN algorithm is I/O optimal.*

Proof 2. *As long as the left edge of (the MBR of) an R-tree node n is not collectively blocked by the DNs of q, the node must be accessed because it may hold DNs. To prove the lemma, we must show that only such nodes are accessed. In our algorithm, a prerequisite to visit a node n is that the DNs found* thus far *do not completely block it from q. To complete the proof, we show that the remaining DNs, i.e., those found* after *visiting n, cannot block any part of its left edge. This is obvious – since R-tree entries and DNs are encountered (i.e., popped from the search heap of the incremental NN search) in increasing order of distance from q, all DNs found after visiting n are further from q and hence cannot possibly block any part of n.*

To efficiently perform the "visibility" check for objects and R-tree nodes encountered during the incremental NN search, we use an adaptation of the segment tree. When the search begins, we initialize an empty segment tree T_E. The first NN of q in the stripe is a DN (by definition it cannot be blocked by any other object), and its y-extent is inserted into T_E. Before our incremental NN search visits any R-tree node, the y-extent of the node's MBR is probed against T_E to detect whether any part of its left edge is exposed to st. If completely blocked, it is pruned (ignored). The same check is performed for every discovered NN r. If r is not (completely) blocked, it is reported as a DN, and its y-extent is inserted into T_E.

In our adapted segment tree, segments are stored only at the leaves of T_E. Those segments that are included in the list of an internal node in a conventional segment tree are instead replicated in the lists of all its descendant leaves. To check an object (or internal R-tree node) with extent $[y_l, y_h]$, we issue a stabbing query on T_E at point y_l. Let n_- be the leaf of T_E that covers y_l. We traverse the leaves to the right of n_- until we reach the leaf n_+ that covers y_h. If any of the segment lists between n_- and n_+ is empty, the object (or R-tree node) passes the test, i.e., a part of its left edge is exposed to st.

DN search in the other three stripes is similar. Note that the y-extents are used in searching the east and west stripes, while the x-extents are used for the north and south ones.

3.3. Complete DN Algorithm

In general, an object may intersect q and/or more than one quadrant or stripe. Essentially, the extent of q, the four stripes and the four quadrants partition the data space into 9 disjoint regions. If an object r intersects more than one region, r is conceptually divided into parts, each falling entirely within one of the regions[3]. The parts are processed independently for the corresponding stripe/quadrant. The final set of reported DNs is the union of the objects that intersect q, and the DNs that are found in the stripes and quadrants. The following discussion refers to our advanced DN algorithm that uses the efficient SDN technique for quadrant search (the case for CNS is similar).

Table 2: Heap contents and result formation

	Heap Contents	**Result**
1	$< N_7, 1 >< N_8, 4 >< N_9, 23 >$	$\emptyset$
2	$< N_2, 1 >< N_8, 4 >$ $< N_1, 13 >< N_9, 23 >$	$\emptyset$
3	$< r_4, 1 >< N_8, 4 >< r_3, 5 >$ $< N_1, 13 >< N_9, 23 >$	$\emptyset$
4	$< N_8, 4 >< r_3, 5 >$ $< N_1, 13 >< N_9, 23 >$	$\{r_4\}$
5	$< N_3, 4 >< r_3, 5 >< N_4, 10 >$ $< N_1, 13 >< N_9, 23 >$	$\{r_4\}$
6	$< r_5, 4 >< r_3, 5 >< r_6, 9 >$ $< N_4, 10 >< N_1, 13 >< N_9, 23 >$	$\{r_4\}$
7	$< N_4, 10 >< N_1, 13 >< N_9, 23 >$	$\{r_4, r_5, r_3, r_6\}$
8	$< r_7, 10 >< N_1, 13 >$ $< r_8, 17 >< N_9, 23 >$	$\{r_4, r_5, r_3, r_6\}$
9	$< r_1, 13 >< r_8, 17 >$ $< r_2, 18 >< N_9, 23 >$	$\{r_4, r_5, r_3, r_6, r_7\}$
10	$< N_9, 23 >$	$\{r_4, r_5, r_3, r_6, r_7, r_1, r_8\}$
11	$\emptyset$	$\{r_4, r_5, r_3, r_6, r_7, r_1, r_8\}$

The search for objects that intersect q, and for DNs in the stripes and quadrants can be performed concurrently, in a single traversal of the R-tree in order to avoid unnecessary I/Os in re-reading R-tree nodes. This is possible, because the search in each quadrant or stripe visits R-tree nodes in increasing distance from q. This means that there can be a single search heap (sorted on $mindist(e, q)$ of encountered R-tree entries e) that serves all the 8 quadrant/stripe searches. During the R-tree traversal, four skyline lists (one per quadrant) and four segment trees (one per stripe) are maintained. Entries (objects) with zero $mindist$ intersect q, and are therefore accessed (reported as DNs) directly. A detailed pseudo-code for this complete, single-traversal DN search is provided in the Appendix.

Since the BBS and stripe DN components of our algorithm are I/O optimal (as proven in [20] and Lemma 2), and since we also avoid re-fetching the same nodes from disk, the overall DN method is I/O optimal, i.e., it performs the minimum possible number of I/Os for the given R-tree structure.

Example 1. *We illustrate our complete DN algorithm with Fig. 4. For simplicity, we only show the NE quadrant, and north and east stripes. Table 2 shows the search heap contents and the DN set in various stages. First, we read the root of the R-tree on S, and push its three entries N_7, N_8, N_9 into the search heap. We then pop the top entry N_7, fetch it from disk and en-heap its children. This process continues until we pop object r_4 in step 3. It falls in the north stripe and we directly insert it into the result set and into the north segment tree T_N. In step 6, we pop r_5 which intersects the NE quadrant and the east stripe. We conceptually partition r_5 and treat each portion as a separate*

[3]Note that this is an implicit partitioning used only for the processing of the specific DN query. It is not persistent, i.e., it does not affect the representation of r on the disk.

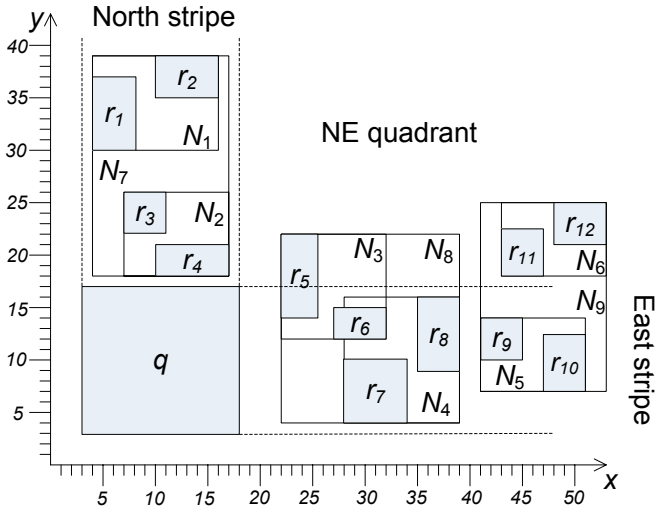


Figure 4: Example of DN search

object in the respective quadrant/stripe. It is a DN in both the NE quadrant and the east stripe, and is thus appended to the result. It is also inserted into the NE skyline and into T_E (the east segment tree). Subsequently popped objects r_3 and r_6 are also DNs, and are inserted into T_N and T_E, respectively. In step 9, object r_2 is popped after r_1 and r_8. It fails the visibility check against T_N (because it is completely blocked by r_3 and r_4) and is discarded. Step 10 pops index entry N_9 which intersects the NE quadrant and the east stripe. We conceptually divide N_9 into two portions. Its portion in the east stripe fails the visibility check against T_E, and its second portion is dominated by the DNs already in the NE skyline (i.e., r_5). Thus, N_9 is ignored (not read from disk). At that stage the heap is empty, and the DN search terminates with result set $\{r_4, r_5, r_3, r_6, r_7, r_1, r_8\}$.

An issue worth mentioning regards the exclusive retrieval region (ERR) of q, described in Introduction. By definition, this region is bounded by the DNs. In a quadrant, the bound is the skyline over the quadrant's DNs. In a stripe, the ERR includes the area that is not blocked by the stripe's DNs. In other words, the ERR can be derived by iteratively subtracting from the entire space the area that is dominated by each quadrant DN and the area that is blocked by each stripe DN (where dominance and horizontal blocking are defined by the specific quadrant or stripe, as explained in Sec. 3.1 and 3.2).

3.4. Arbitrary Object Shapes

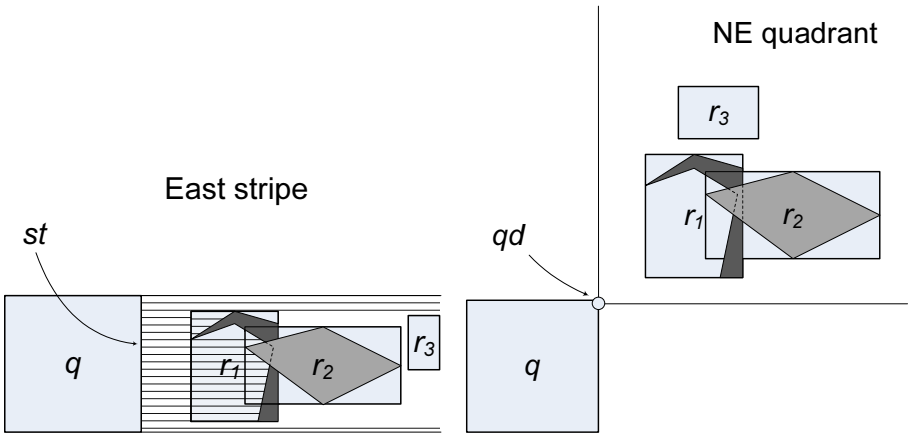
So far we have focused on rectangular objects. Our techniques, however, extend easily to arbitrary object shapes.

Stripe DNs. The stripes are processed similarly to rectangular objects. The difference is that the object lists of the segment tree (used for a specific stripe) hold both the MBR and the exact geometry of each discovered DN. In checking the visibility of an R-tree node, the exact geometries in the object lists are only taken into account when their MBRs intersect that of the R-tree node in question. Visibility check for a leaf node entry (i.e., MBR of a data object) proceeds similarly, except that if the MBR intersects (the actual geometry of) a DN, the object's exact geometry must also be fetched to complete the check. Consider, for example, Fig. 5(a) and DN search in the east stripe. Assume that r_1 is the only DN found so far. The next encountered node is r_2, which overlaps the MBR of r_1. The exact geometry of r_1 is also overlapping the MBR of r_2, and we can only determine whether the latter is a DN by fetching its own exact geometry too. The comparison reveals that a part of r_2 is unblocked and therefore it is a DN. On the other hand, we may infer that r_3 is not a DN without fetching any exact object geometries, because its MBR does not overlap that of r_1, and it is fully blocked by it.

Quadrant DNs. The quadrant search traverses the R-tree of S following the BBS strategy as normal. An internal R-tree node is loaded only if it is not dominated by any existing skyline object. The MBR of a skyline object can be used for quick dominance check. If the R-tree node does not intersect the MBR, dominance (or not) is definitively decided based on their closest corners to q. If there is overlap, the skyline object's exact geometry must be taken into account.

When a leaf node entry is popped (i.e., the MBR of a data object), we check whether the MBR is dominated by any object already in the skyline. If not, we fetch the exact geometry of the corresponding object from the disk and push it into the heap with its *actual* minimum distance from q as key value. If the object is popped subsequently, we check (using its exact geometry) whether it is dominated by any existing skyline object. If not, it is added to the skyline and to the DN set.

In Fig. 5(b), for instance, assume that we have discovered one DN so far, r_1. We can infer that r_3 is not a DN using only MBR information; the MBR of r_3 does not overlap that of r_1 and is also dominated by it. In contrast, when we encounter r_2 we cannot make safe conclusions based on its MBR (because it overlaps r_1); therefore, we fetch its exact geometry, and enheap it with its actual distance as key. When the latter is popped



(a) Stripe DN (b) Quadrant DN

Figure 5: DN search for arbitrarily shaped objects

again, we compare its exact geometry with that of r_1 and determine that a part of it is not dominated. Thus, r_2 is also a DN. A detail regards the dominance check between exact geometries represented as polygons: object r_2 is not dominated by r_1 because one of its polygon vertices (the left-most in this case) is not dominated by any vertex of r_1.

4. K-Direct Neighbor Search

An intuitive way to compute a broader spectrum of "neighbors" is to relax the DN condition to allow up to some number of intervening objects in the query window (in addition to q and each of its DNs). Specifically, an object $r \in S$ is a K-DN of q if and only if there exists a query window that intersects q, r, and no more than $K-1$ other objects. The K-DN query is a generalization of plain DN (the latter corresponds to $K=1$).

Stripe K-DNs. Following a similar definition to plain DN, an object r is a K-DN of q in a stripe, say the east, if there exists a horizontal ray from any point on q's right edge st to a point on r's left edge that cuts through fewer than K other objects. In Fig. 3(a), for example, the 2-DNs of q in the east stripe include the 1-DNs r_1, r_2, r_3, plus objects such as r_4 and r_6 which can be "reached" from q via a horizontal ray that cuts across exactly one other object. In the case of r_4 the intervening object is r_2.

Identifying K-DNs in a stripe is similar to plain DN. Take the east stripe as example. A stripe-constrained, incremental NN query is posed at st (i.e., the right side of q), examining objects r in increasing $r.x_l$ order. Whenever a K-DN is found, it is inserted into the segment tree T_E. When the constrained NN search considers whether to visit (i.e., read from disk) an internal node of the R-tree, we probe T_E and examine all its leaf nodes that overlap the y-extent of the R-tree node. If at least one of them has fewer than K objects in its list, the R-tree node is visited. Data objects r discovered in the NN search are reported as K-DNs if they pass the same test, or disqualified otherwise. The stripe DN operation is I/O optimal, with the constrained NN search following the best-first paradigm [9] in conjunction with pruning R-tree nodes that fail the segment tree test. The proof is similar to Lemma 2.

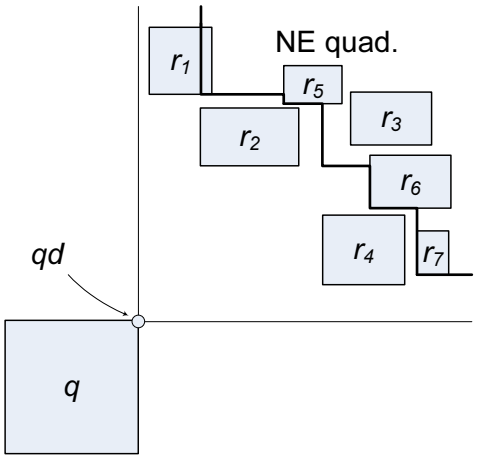


Figure 6: 2-DNs in NE quadrant (2-skyband)

Quadrant K-DNs. The definition of MIA and Observation 1 extend to $K > 1$. Let r be an object in a quadrant. As established in Sec. 3.1, any query window that intersects both q and r must completely envelop the MIA of r. Thus, the necessary and sufficient condition for r to be a K-DN is that its MIA intersects fewer than K other objects.

CNS can be used to retrieve such objects using a K-tier ($90°$-constrained) NS query. However, this inherits the deficiencies identified in Sec. 3.1, namely that a superset of the actual K-DNs is returned, which entails unnecessary I/Os and a post-processing (filtering) step. As we show in the experiments, the problem is exacerbated as K increases, because a larger fraction of the NSs are not DNs.

On the other hand, Lemma 3 extends SDN to $K > 1$, enabling I/O optimal processing. Let qd be the corner of q that anchors the quadrant to be processed. Recall that the K-skyband is a generalization of the skyline that includes objects that are dominated by fewer than K others.

Lemma 3. *The K-DNs of q in a quadrant are exactly the K-skyband objects in the quadrant, with qd as origin and each object represented by its closest corner to qd.*

Proof 3. *Let $r \in S$ be an object in the quadrant of qd. The necessary and sufficient condition for r to be a K-DN is that its MIA intersects fewer than K other objects. Any object r' in the quadrant of qd that overlaps r's MIA has, by definition, its closest corner to qd inside the MIA, and vice versa. Hence, r is a K-DN if and only if its MIA contains fewer than K closest-to-qd corners of other objects, i.e., it is dominated by fewer than K objects.*

Fig. 6 illustrates a 2-DN search in the NE quadrant of q. The bold staircase line corresponds to the 2-skyband boundary, implying that anything that does not touch this boundary and lies further NE from it (i.e., further to the right and top), is dominated by at least 2 objects. The 2-skyband comprises objects whose closest corner to q either falls on the staircase line (like r_6) or lies SW from it (like r_1). The 2-DN set (equivalently, the 2-skyband) includes r_1, r_2, r_4, r_5, r_6, r_7. Object r_3 is not a 2-DN because it is dominated by r_2 and r_4. The K-DNs in each quadrant can be computed with the I/O optimal K-skyband BBS algorithm of [20].

Similar to the complete plain DN case in Sec. 3.3, all the K-DNs (in the stripes or quadrants, or those intersecting q) can be retrieved in a single traversal of the R-tree that indexes S, yielding an overall I/O optimal solution. Furthermore, adopting the branch-and-bound paradigm in the SDN-based K-DN method for the stripes and the quadrants lends two desirable properties, namely, that the method is *progressive* and *incremental*. Progressive implies that DNs can be reported as they are discovered, without waiting for the algorithm to terminate. The incremental nature of our method implies that there is no need to fix K in advance. Should the user originally specify a K value that turns out to be too small, the retrieval of DNs for a larger K does not need to start from scratch, but can resume from where the previous (smaller K) search stopped. Of course, to support incremental processing, pruned R-tree nodes and rejected objects cannot be discarded, but must be kept (sorted on increasing distance from q) in anticipation of an increase in K.

Example 2. *Consider the example in Figure 4, and assume q retrieves K-DNs with $K = 2$. The K-DN search differs from DN search in the way it prunes non-result objects. We focus on pruning and omit other details for simplicity. The search is exactly the same as before until step 10 in Table 2, where r_2 is kept as a 2-DN because, while using the north segment tree T_N to verify the visibility of r_2, we find that only r_4 is blocking it from q. Next we have to check N_9, since parts of it are blocked by only one object according to the east segment tree T_E. Children r_9 and r_{10} of N_5 pass the check, since none of them is separated by two or more objects from q. In the NE quadrant, r_{11} is a 2-DN whereas r_{12} is not, because the MIA of r_{12} intersects r_5 and r_{11}. Now the heap becomes empty, and the search terminates with result set $\{r_4, r_5, r_3, r_6, r_7, r_1, r_8, r_2, r_9, r_{11}, r_{10}\}$.*

5. All-Direct Neighbor Query

Another extension to DN search is the *all-DN* (ADN) query. Given a set of objects S, an ADN query computes for each object in S all its DNs. A straightforward way to process ADN queries is to apply a plain DN search using, in turn, each object $r \in S$ as source. Clearly, this approach is inefficient. Performance can be improved if the entire R-tree is loaded in memory to avoid multiple reads from the disk (since all of its contents need to be accessed anyway). However, the CPU cost remains a major drawback. In the following we describe an algorithm for ADN processing that significantly reduces the processing time of individual DN retrievals, by sharing computations among them.

5.1. Fundamental Properties of DNs

We begin with observations/properties of DN relationship.

Observation 2. *The DN relationship is symmetric.*

By definition, if object r' is a DN of another object r, there exists a query window intersecting only r' and r, regardless of whether r' lies in a stripe or quadrant of r. Thus, r is a DN of r' too.

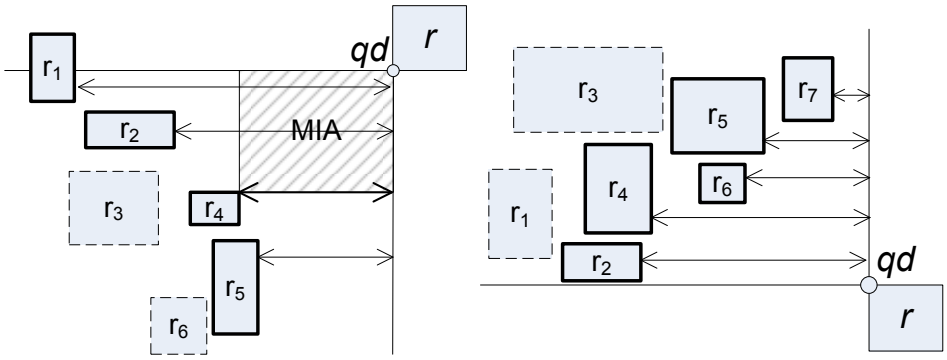
According to Observation 2, the ADN query requires finding the DNs in only two quadrants and two stripes of each object instead of four. For example, we may consider only the NW and SW quadrants, and only the west and south stripes. The rationale is that if another object r' is a DN with respect to, say, the NE quadrant of r, then r' will definitely identify r as a DN in its SW quadrant.

Observation 3 formulates an important property of quadrant DNs. We take the SW and NW quadrants as an example, and illustrate in Fig. 7.

Observation 3. *Given a rectangle r and its set of DNs in the SW (or NW) quadrant, there exists a portion on the right side of each of these DNs that is horizontally unblocked from the vertical edge of the quadrant.*

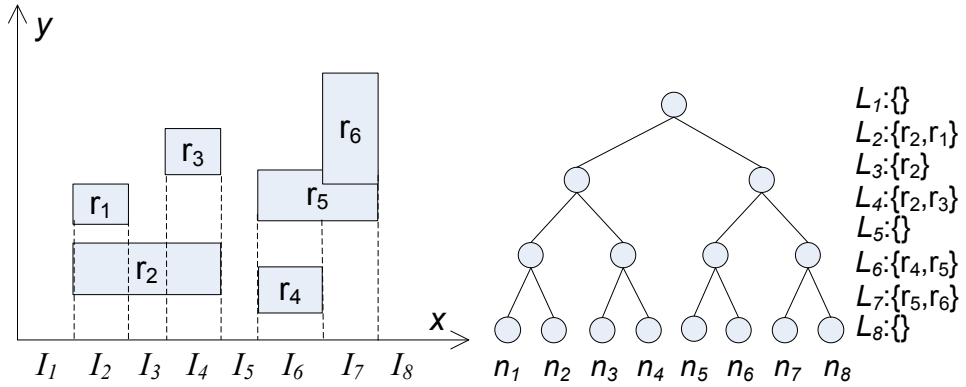
Fig. 7(a) illustrates that the DNs in the SW quadrant of r are all visible via horizontal rays shot from the vertical quadrant edge (i.e., the vertical half-line with qd as initial point). The DNs are shown with solid border, and the horizontal rays are shown as arrows. Observation 3 follows from Observation 1. DN r_4, for instance, is definitely "visible" by some horizontal ray, because its MIA (and therefore the bottom edge of the MIA) intersects no other object. The converse, however, is not true; in other words, not all objects visible via horizontal rays are DNs. The two objects with dashed border (r_3, r_6) are both visible but are not DNs. The situation in the NW quadrant is similar (see Fig. 7(b)).

Our ADN algorithm, presented next, builds on the above observations to achieve efficient processing.



(a) DNs in SW quadrant (b) DNs in NW quadrant

Figure 7: Properties of quadrant DNs



(a) Objects and x-intervals (b) T_x and object lists

Figure 8: Segment tree on the x-extents of objects

5.2. The All-DN Algorithm

Our approach aims to maximize *computation sharing* among DN retrievals of the objects in S. It achieves this by using two segment trees, T_x and T_y, and performing a sweep of the data space from left to right. Based on Observation 2, we compute for each $r \in S$ its DNs in the SW and NW quadrants, and the west and south stripes.

Fetching all the objects[4] from disk, we first construct a segment tree T_x on their x-extents. Each leaf node n_k^x in T_x is associated with a list L_k^x of objects whose x-extents overlap the interval corresponding to n_k^x. Objects stored in L_k^x are maintained in ascending order on their lower y-values. Fig. 8 illustrates a segment tree on the x-extents of 6 objects. Fig. 8(a) shows on the x-axis the intervals I_k^x associated with each leaf, and Fig. 8(b) presents the tree structure and the object lists L_k^x of the leaves. Note that the superscript x indicates the dimension indexed by the segment tree, but is omitted from the figure for clarity.

In constructing T_x, we do not insert objects one by one. Instead, T_x is bulk-loaded. After sorting the objects in S on their lower x-values, we use the resulting $2|S| + 1$ x-intervals (and their object lists) as leaf nodes. Then, we build the segment tree bottom-up.

After constructing T_x, we scan from its leftmost leaf node to the rightmost. For each leaf n_k^x, we consider the objects in its list in ascending lower y-value and compute their DNs (in the SW and NW quadrants, and in the west and south stripes). After processing all the objects in the list, we proceed to the next leaf node, until all the leaves are visited. The details are as follows.

South stripe DNs. Suppose $\{n_1^x, n_2^x, ..., n_{2|S|+1}^x\}$ is the sequence of leaf nodes in T_x. Starting from the leftmost leaf's list and moving to the right, we identify as DNs (in the south stripe) every pair of successive objects $\langle r_i, r_{i+1} \rangle$ in the list. Correctness is obvious, since by definition there is no object between r_i and r_{i+1} in the x-interval of the corresponding leaf. Consider, for example, Fig. 9. The list for node n_k^x contains (among others) objects r_{10} and r_{11}. Due to the sorting of the list on lower y-value, r_{10} and r_{11} are placed consecutive to each other and, hence, they are correctly reported as DNs.

The remaining DNs (west, SW, NW) are also discovered during the aforementioned left-to-right scanning of T_x. This task is facilitated by a second segment tree T_y, similar to T_x, which however (i) is built on the y-extents of objects, and (ii) is incrementally populated. T_y is empty initially. On encoun-

[4]Note that any ADN algorithm has to read S from the disk. Also, the size of T_x is manageable, as we show in experiments. In seriously memory-constrained systems, a disk-based segment tree can be used [29].

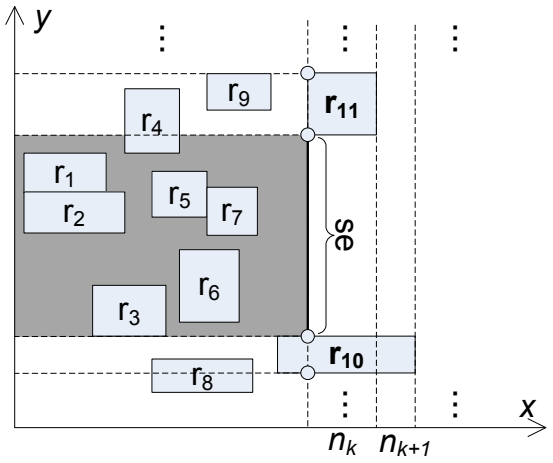


Figure 9: Finding west, SW and NW DNs of r_{11}

Leaf of T_y	Object list
n_1	{r8}
n_2	{r8, r10}
n_3	{r10}
n_4	{r3}
n_5	{r3,r6}
n_6	{r6}
n_7	{}
n_8	{r2,r7}
n_9	{r2,r5,r7}
n_{10}	{r1,r5}
n_{11}	{r1}
n_{12}	{r4}
n_{13}	{r4,r9}
n_{14}	{r9}

(a) Before insertion

Leaf of T_y	Object list
n_1	{r8}
n_2	{r8,r10}
n_3	{r10}
n_4	{r3}
n_5	{r3,r6}
n_6	{r6}
n_7	{}
n_8	{r2,r7}
n_9	{r2,r5,r7}
n_{10}	{r1,r5}
n_{11}	{r1}
n_{12}	{r4}
n_{13}	**{r4,r11}**
n_{14}	**{r4,r9,r11}**
n_{15}	**{r9,r11}**

(b) After insertion

Figure 10: Inserting r_{11} into T_y

tering an object r in some T_x node for the first time, we insert it into T_y (according to its y-extent). The objects in the leaves of T_y are sorted in ascending order on their upper x-value. In the following we explain how T_y enables the retrieval of the remaining three types of DNs and helps in sharing computations in the ADN retrieval.

West stripe DNs. Whenever an object r is inserted into a leaf node n_k^y of T_y, we immediately identify the last object in n_k^y as a DN with respect to the west stripe of r. This is because that object has the largest upper x-value among all the objects to the left of r, in the y-interval that corresponds to n_k^y. Continuing the example in Fig. 9, Fig. 10(a) and 10(b) illustrate the leaf node lists in T_y before and after the insertion of r_{11}. For clarity, the superscript y is omitted from the leaf names, and leaves irrelevant to the example are not shown. The insertion creates a new leaf in T_y, and r_{11} is appended in the object lists of $n_{13}^y, n_{14}^y, n_{15}^y$. The rightmost objects in these lists are r_{11}'s west DNs (r_4 in L_{13}^y, along with r_9 in L_{14}^y and L_{15}^y).

NW and SW DNs. T_y is used for efficient NW and SW DN search too. Without loss of generality, assume that no pair of objects have exactly the same lower or upper x-value. Consider again the scanning of leaf nodes in T_x in the process of retrieving the south and west DNs. For each leaf n_k^x, the left bound of its associated interval is due to either the lower or upper x-value of an object. In Fig. 9, for instance, the left bound of n_k^x is due to the *lower* x-value of r_{11}, whereas the left bound of n_{k+1}^x is due to the *upper* x-value of r_{11}. For leaf nodes of the former category, we compute the NW and SW DNs of the responsible object; i.e., when considering n_k^x we compute the NW/SW DNs of r_{11}, whereas no NW/SW processing is needed for n_{k+1}^x.

We now focus on n_k^x and the SW processing for r_{11}. By definition, all the SW DNs of r_{11} are inside the gray area in Fig. 9, i.e., inside the region $[0, r_{11}.x_l][r_{10}.y_h, r_{11}.y_l]$. Note that this "stripe" is delimited by r_{10}, the preceding object in the list of n_k^x. Since the left bound of n_k^x is attributed to $r_{11}.x_l$ and r_{10} is in L_k^x too, it holds that $r_{10}.x_l < r_{11}.x_l$, i.e., there is a portion of r_{10} protruding to the left of n_k^x. This portion of r_{10} disqualifies any object lower than the gray area from being a SW DN of r_{11}.

To identify the DNs in the gray area, we use Observation 3 as a filtering step to produce an (inclusive) list $\mathcal{R}$ of candidate SW DNs. Specifically, we utilize T_y to retrieve those objects in the gray area that are horizontally unblocked from the left of line segment se (shown in the figure). The process is similar to retrieving an object's west DNs using T_y. Fig. 10(b) shows the state of T_y when processing r_{11}. The leaves that overlap se (on the y-extent) are n_4^y up to n_{12}^y. $\mathcal{R}$ is formed by collecting the last (i.e., rightmost) objects in the lists of these T_y leaves, i.e., $\mathcal{R} = \{r_4, r_1, r_5, r_7, r_6, r_3\}$. The leaves of T_y are scanned from n_{12}^y towards n_4^y (i.e., from higher y-values to lower ones), leading to an inherent sorting of $\mathcal{R}$ according to upper y-values. This sorting facilitates subsequent refinement.

The refinement step eliminates non-DN entries from $\mathcal{R}$ based on Lemma 1, i.e., by finding the skyline objects in it. Due to the inherent ordering in $\mathcal{R}$, the time needed for skyline processing is linear to $|\mathcal{R}|$. We scan $\mathcal{R}$ from the first to the last object, i.e., in decreasing upper y-value. The first object, r_4, must be a SW DN as its highest y-value means that it cannot be dominated by any other candidate. For each subsequent candidate r in $\mathcal{R}$ to be a skyline object (i.e., a DN), it must have an upper x-value larger than that of the *last identified DN*. Note that this single comparison saves considerable computations compared to checking r for dominance against all the skyline objects found. In our example, the second candidate, r_1, is not a DN because its upper x-value is lower than that of r_4. The third candidate, r_5, passes this check against r_4 and becomes the last reported DN. Thus, the fourth candidate, r_7, is checked against r_5 *only* (as opposed to the entire skyline). Object r_7 is reported as a DN, and subsequently disqualifies r_6 and r_3, leading to r_4, r_5 and r_7 being the final SW DNs of r_{11}.

A symmetric filter-and-refine procedure is used to retrieve the NW DNs of r_{11}. The difference is that the corresponding (gray) search area is refined by r_{12}, the next object in the list

Algorithm 1: ADN

Input: a set S of objects
Output: set $\mathcal{R}_{\mathcal{DN}}$ of DN pairs in S

```
1  build segment tree T_x on x-extents of objects in S;
2  N_x ← sequence of leaf nodes of T_x;
3  initialize an empty segment tree T_y;
4  R_DN = ∅;
5  for i = 1 to |N_x| do
6      let L_i^x be the object list of node n_i^x;
7      for each object r_j in L_i^x do
8          append the pair ⟨r_{j-1}, r_j⟩ to R_DN;
9          if left bound of n_i^x is due to r_j.x_l then
10             R_DN = R_DN ∪ SearchGrayArea(T_x, T_y, r_j);
11             insert r_j into T_y according to its y-extent;
12             for each leaf n_k^y covered by [r_j.y_l, r_j.y_h] do
13                 append the pair ⟨r_j, r'⟩ to R_DN, where
                   r' is the last object in the list of n_k^y;
14 return R_DN;
```

of n_k^x. If r_{11} is the last object in L_k^x, the search area extends upwards to the boundary of the data space. Also, the traversal of T_y nodes that fall in the search area is performed in increasing y-order, leading to an $\mathcal{R}$ that is sorted on the lower (instead of the upper) y-value of the candidate objects.

Note that the filtering and refinement steps (for either SW or NW DNs) have a cost linear to the size of $\mathcal{R}$, since no sorting is required in the skyline computation. The latter is taken care of by the structure of T_y. There is thus only a one-time cost in inserting each encountered object into T_y which, once spent, is utilized by all the subsequently considered objects. This avoids many unnecessary computations (compared to independent DN retrievals for every $r \in S$). Finally, the algorithm extends trivially to all-K-DN processing, following a methodology similar to Sec. 4.

The detailed ADN algorithm is given in Algorithm 1. Line 8 finds DN pairs from the north and south stripes, line 10 searches for SW and NW DNs in the gray area (using Algorithm 2), whereas line 13 identifies the west and east stripe DNs.

Algorithm 2: SearchGrayArea

Input: T_x, T_y, an object r_j
Output: NW and SW DNs of r_j

```
1  R_SW = ∅;
2  issue a stabbing query on T_y at r_j.y_l;
3  let n_+ be the resulting leaf node in T_y;
4  scan from n_+ towards the leaves with smaller y-values
   until reaching the boundary of a leaf n_− whose last
   object's upper x-value satisfies x_h > r_j.x_l;
5  let R be the sequence of the last objects in the lists for
   the leaves between [n_+, n_−];
6  while R is not empty do
7      remove the first object r' from R;
8      let r'' be the last added DN to R_SW;
9      if r'.x_h > r''.x_h then append ⟨r_j, r'⟩ to R_SW;
10 R_NW = ∅;
11 issue a stabbing query on T_y with r_j.y_h;
12 let n_− be the resulting leaf node in T_y;
13 scan from n_− towards the leaves with greater y-values
   until reaching the boundary of a leaf n_+ whose last
   object's upper x-value satisfies x_h > r_j.x_l;
14 let R be the sequence of the last objects in the lists for
   the leaves between [n_−, n_+];
15 while R is not empty do
16     remove the first object r' from R;
17     let r'' be the last added DN in R_NW;
18     if r'.x_h > r''.x_h then append ⟨r_j, r'⟩ to R_NW;
19 return R_SW ∪ R_NW;
```

6. DN Search in Higher Dimensions

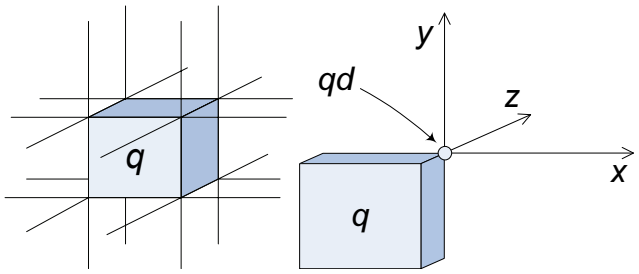
Our methodology extends beyond two dimensions while retaining its I/O optimality. In this section we discuss the three-dimensional case (i.e., $d = 3$) and then generalize to more dimensions. Prior to that, however, we establish that DN search is meaningful primarily in low-dimensional spaces, and justify our focus on this setting.

6.1. Effect of Dimensionality

A hyper-rectangle q in d dimensions has 2^d vertices – in two-dimensional terms, vertices correspond to corners. Equivalently, each of the vertices defines a space partition. In a way similar to quadrants, the DNs in each of these 2^d partitions coincide with the skyline objects (the details of why this is the case are discussed later in this section). On the other hand, it is known that the number of skyline points increases exponentially with dimensionality [30]. Since the DN set is a superset of 2^d skylines (each of exponential cardinality with d), the number of DNs increases at least exponentially with d. In other words, in high dimensionality the DN set includes a large fraction of the dataset S. This limits the usefulness of the query, since the very motivation of the DN problem is to identify a small subset of S as immediate competitors of q. Its diminishing selective power with d suggests that it is more meaningful in low dimensional spaces (a feature common in many queries, like skylines [30], NNs [31], etc). Furthermore, in practice, window queries in applications like propertyguru.com.sg and kayak.com (mentioned in Introduction) typically do not involve more than three or four dimensions. Notwithstanding this, our DN processing methodology extends to $d > 2$ and remains I/O optimal (regardless of dimensionality).

6.2. Processing in Three Dimensions

In three dimensions, the source and data objects are three-dimensional boxes, and S is indexed by a 3-D R-tree. The DN relationship is expressed in terms of 3-D window queries, i.e., two objects are DNs if and only if they can be exclusively inter-



(a) Partitions (b) Vertex partition at qd

Figure 11: DN search in three dimensions

sected by some axis-parallel box. The geometry of the source object q includes 8 vertices, 6 faces, and 12 edges. Each of these elements defines a partition, which leads to 27 partitions in total (including box q itself) as shown in Fig. 11(a).

The treatment of vertices is similar to that of quadrants in two dimensions. Consider the partition defined by vertex qd in Fig. 11(b). The MIA of an object r that falls in this partition is the three-dimensional box with a diagonal from qd to the vertex of r that is closest to qd. Lemma 1 extends trivially to three dimensions; the DNs in a vertex partition are the skyline objects in this partition with the vertex (i.e., qd) as origin. BBS can be used to compute the 3-D skyline in an I/O optimal way [20]. Note that CNS is no longer applicable, because there is currently no NS algorithm for more than two dimensions.

Faces are handled like stripes in 2-D, the difference being that a spatial access method (e.g., a main-memory 2-D R-tree) is used instead of a segment tree for visibility check. An incremental NN search is initiated at the face, directed outside of q. Denote the face as st and its 2-D R-tree as T_{st}. The projections of encountered DNs on st (projections are necessarily rectangular) are inserted into T_{st}. The incremental NN search in the index of S prunes nodes and ignores objects whose projections on st are entirely covered by DN projections already in T_{st}.[5]

In Fig. 12(a), the rectangles on face st are the projections of three objects (numbered according to the subscripts of the objects). r_1 is the first NN of st, and is thus a DN. Object r_2 is not a DN because its projection is fully covered by that of r_1. If r_2 were the MBR of an index node, it would be pruned. On the other hand, r_3 is a DN because it is not fully blocked by r_1. At this stage, T_{st} includes the projections of r_1 and r_3 (i.e., of the DNs found so far).

Edges require special treatment. Each edge ed has two *fixed* dimensions and one *variable*. In Fig. 12(b), the x and y dimensions are fixed (every point on ed has the same x and y coordinates) while z is variable. The MIA of an object r in the partition of ed is the box defined by ed and the closest edge of r that is parallel to ed. Objects that could potentially disqualify r from being a DN must first of all dominate it in the fixed dimensions (x and y). Such a dominating object $r' \in S$ disqualifies the part of r that overlaps with its z extent (if there is any overlapping). In Fig. 12(b), for instance, r_1 dominates r_2 in the subspace of fixed dimensions (i.e., the $x : y$ plane). In the variable dimension (z), the z-extent of r_1 only partially covers that of r_2 (see their projections on ed), and disqualifies only that part of r_2. The remaining part of r_2 may still be exclusively intersected (along with q) by a three-dimensional window query, i.e., r_2 is a DN.

Formally, an object r in the partition of edge ed is a DN if and only if the objects that dominate it in the fixed dimensions do not collectively cover its extent in the variable dimension. The proof resembles that of Lemma 1 and is omitted. We adapt BBS to perform DN search in the partition of ed. The algorithm proceeds like a skyline computation, i.e., with an incremental NN search at ed. A node is visited (or an object included in the DN set) if the DNs found so far that dominate it in the fixed dimensions, do not collectively cover its entire extent in the variable dimension. The search inherits the I/O optimality of BBS algorithm.

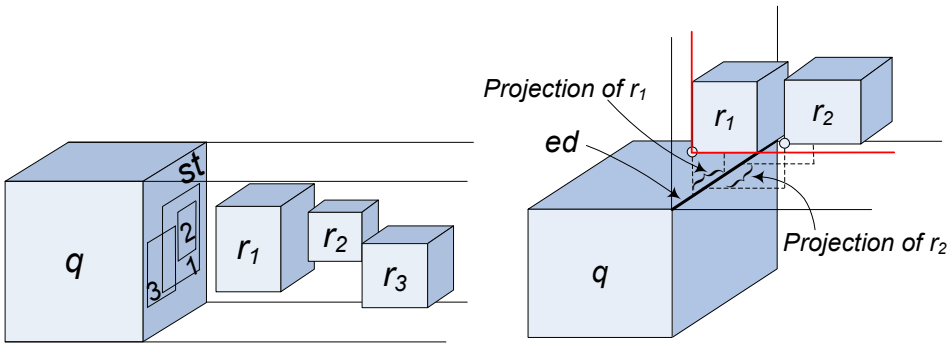
The DN search inside q and in the partitions defined by vertices, faces and edges can be completed in a single R-tree traversal, similar to Sec. 3.3, which guarantees overall I/O optimality.

6.3. Beyond Three Dimensions

The geometry of a d-dimensional hyper-rectangle q includes d types of elements; e.g., in two dimensions we have vertices and edges (defining quadrants and stripes), while in three dimensions we additionally have faces. Every element has m fixed dimensions and $d - m$ variable ones, where $1 \leq m \leq d$. When $m = d$ the element is a vertex, and a skyline search retrieves exactly the DN set in the corresponding partition. In all other cases ($m < d$), an object r can only be disqualified by objects that dominate it *in the fixed dimensions*. Each of these dominating objects disqualifies the portion of r that overlaps with it in the variable dimensions. Note that this applies also to the $m = 1$ case (e.g., stripes in two dimensions, or faces in three) where there is a single fixed dimension, and dominance degenerates to closeness to q (thus the incremental NN search on the fixed dimension that we used previously for stripes and faces). Each of these partitions can be processed with BBS where a node or object is pruned if the already discovered DNs that dominate it in the fixed dimensions, collectively cover its entire extent in the variable dimensions. The DN search for all the partitions (and for DNs inside q) can be performed in a single R-tree traversal, thus guaranteeing I/O optimality.

Regarding K-DN search in higher dimensions, it follows the principles presented in this section. Under the same generalized definition of dominance (which takes into account fixed and variable dimensions), K-DN search disqualifies/prunes R-tree nodes and data objects that are dominated by at least K result objects found so far. All-DN extension is similar to Sec. 5, where all objects are loaded into a segment tree on one dimension, and an (incrementally populated) R-tree on the remaining dimensions plays the role of T_y to facilitate dominance checking.

[5] To perform this check, when considering an object r (or an index node), we process a window query in T_{st} to retrieve all the DN projections (rectangles) that overlap with that of r. After subtracting these rectangles from the projection of r using a polygon clipping algorithm [32], we check whether there is a residual. If so, the object (or node) passes the test.



(a) Face partition at st (b) Edge partition at ed

Figure 12: DN search in face and edge partition

7. Empirical Evaluation

In this section, we evaluate our algorithms using real and synthetic data. For the two-dimensional experiments, we obtained three real datasets from the R-tree portal[6], namely LB, TCB and CA. LB and CA contain 53K and 2.2M MBRs, respectively, of roads and streets in Long Beach and California. TCB contains 556K MBRs of residential blocks in four American states. The distribution in CA is highly skewed, whereas LB and TCB are more evenly spread out. The MBRs in each of them are treated as data objects in the experiments. To control the dataset cardinality, we also produced synthetic data with the generator of Theodoridis et al. from the R-tree portal[7]. The generated rectangles are distributed in a $[0, 10000][0, 10000]$ space uniformly, and have a side-length of 10 units on average.

Table 3: Plain DN results

Dataset	# of I/Os		CPU time (msec)	
	SDN	CNS	SDN	CNS
LB	29	61	24	306
TCB	17	129	61	313
CA	22	174	67	400
Synthetic	55	112	74	226

Table 4: Number of DNs and NSs

Dataset	Stripe DNs	Quad. DNs	NSs
LB	12	6	72
TCB	14	3	53
CA	11	10	69
Synthetic	11	13	126

Our evaluation also includes higher-dimensional experiments. For these, we use the HOTEL dataset (from *hotelsbase.org*), which contains 418,843 hotel records with four attributes, namely stars, price, number of rooms, and number of facilities. We normalized the dataset to a [0,10000]^4 space. Since hotel records correspond to points, we extended them to hyper-rectangles with average side-length of 10 units.

All the datasets are indexed with R*-trees [6], using a page size of 4KBytes. The algorithms are implemented in C++, and run on a Ubuntu machine with a 2GHz Intel Core Duo CPU and 2 GBytes of main memory.

7.1. Comparison with Related Query Types

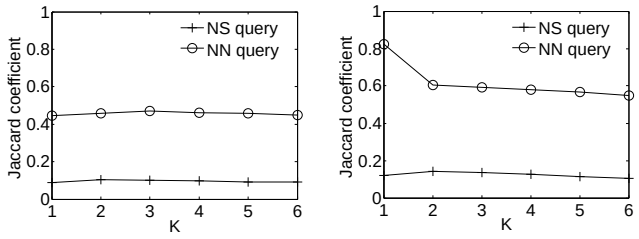
Before investigating processing performance, it is essential to quantify how different the results of this new query (DN) are from those of related query types, namely, nearest neighbors (NN) and nearest surrounders (NS). First, we process K-DN queries for $K = 1$ to $K = 6$ and record the result sets (in this experiment we focus merely on the composition of the result). Then, we process NN and NS queries at (the centroid of) the same source objects as DN search[8]. In the NN case, we produce as many NNs as the cardinality of the corresponding K-DN set. In the NS case, we retrieve K-tier NSs for the same K value as the respective K-DN set.

To compare the result sets, we compute the Jaccard similarity coefficient, a standard means to measure similarity between sets [33]. The Jaccard coefficient of two sets A and B is defined as the cardinality of their intersection divided by the cardinality of their union, i.e., as $\frac{|A \cap B|}{|A \cup B|}$. Fig. 13 plots the Jaccard coefficient of NN and NS results with DN sets for different K values.

There are three key observations. First, NN sets have a higher similarity to DNs than NSs. A reason for this is that we "favor" NN by producing the same number of NNs as DNs (this number could not be known in advance, unless we run a K-DN query first); on the other hand, K-tier NS sets have very different (i.e., larger) cardinality than K-DNs. Second, the more oblong the objects in a dataset, the more the NN and NS results deviate from the DNs. This explains why in LB, CA, and Synthetic the Jaccard similarity is considerably smaller that TCB – objects in TCB have an average aspect ratio of 1.908, whereas the average aspect ratios in LB, CA, and Synthetic are 6.631, 5.920, and 3.857, respectively. The final observation is that similarity generally drops for larger K. This is expected,

[6]http://www.rtreeportal.org
[7]http://www.rtreeportal.org/software/SpatialDataGenerator.zip

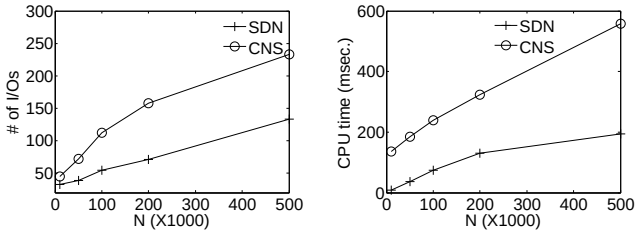
[8]Recall that there exist NS methods only for point sources in two-dimensional domains.



(a) LB (b) TCB

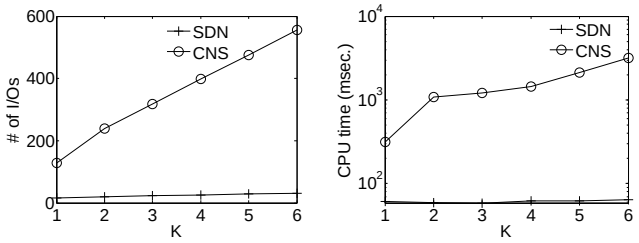
(c) CA (d) Synthetic

Figure 13: Jaccard coefficient of NN and NS sets w.r.t. DN results



(a) # of I/Os (b) CPU time

Figure 14: Plain DN, effect of N (Synthetic dataset)



(a) # of I/Os (b) CPU time

Figure 15: K-DN search, effect of K (TCB dataset)

because as K increases, so do the sizes of the compared result sets, and therefore the differences between the semantics of the queries become more pronounced.

7.2. Experiments in Two Dimensions

Plain DN results. We first consider plain DN search in two dimensions. We compare our SDN-based algorithm versus the baseline CNS-based method, denoted in the charts as SDN and CNS, respectively. For fairness, we enhanced the latter with a single-traversal optimization (similar to that in Sec. 3.3) to avoid multiple reads of the same R-tree nodes. Moreover, its NS search component uses the most efficient algorithm proposed in [10] (termed *Sweep* in that work). Performance is measured in terms of I/O cost and CPU time. Note that due to the single traversal feature of both algorithms, the existence or not of a cache does not affect the I/O cost (because the R-tree nodes are accessed at most once).

For each line in Table 3, we use one of the four datasets as S and choose the source object at random among its rectangles. In this experiment, the cardinality of Synthetic is set to 100K. Every reported measurement is the average over 100 queries (at different source objects). CNS incurs 2 to 8 times more I/Os (in LB and CA, respectively). The reason is that the majority of quadrant NSs are false positives, i.e., they are not actually DNs. Table 4 illustrates the average number of stripe DNs, quadrant DNs, and NSs. The number of NSs is an order of magnitude larger than actual quadrant DNs, which implies a large false positive ratio in CNS and translates to a considerable number of (unnecessary) I/Os. Furthermore, turning again to Table 3, CNS requires significantly more CPU time. The reason is not only that there are too many NSs, but also that angular search in CNS is more complex than skyline computation.

In Fig. 14, we examine the effect of dataset cardinality N on the performance of CNS and SDN. We use synthetic data to effectively vary N from 10K to 500K. The results show that although both algorithms incur proportionally higher costs with a larger N, SDN continues to maintain a substantial lead over CNS. At $N = 500$K, for instance, SDN incurs around half the I/O cost of CNS, and one third the CPU time.

K-DN results. We evaluate SDN and CNS for K-DN search. For brevity, we present results only for the TCB dataset (the trends and relative performance for the other datasets are similar). In Fig. 15, we vary K from 1 to 6 and measure the I/O cost and CPU time of the algorithms. Each plotted value is the average over 100 randomly chosen source objects. CNS incurs one to two orders of magnitude more I/Os than SDN (in Fig. 15(a)). The gap in CPU time is even wider (in Fig. 15(b)); CNS requires several seconds to process a DN query, whereas SDN spends less than 100msec in all cases. SDN's superior performance is due to its I/O optimal and CPU-efficient BBS search. In contrast, CNS wastes I/O and CPU time on the numerous false positives (NSs that are not DNs). The problem is exacerbated as K increases from 1 to 6 (corresponding to 1-tier to 6-tier NS retrieval); the average number of NSs found by CNS grows from 53 to 6593 per query, among which there are only 3 to 10 actual DNs.

Next, we repeat the previous experiment and examine the space requirements in the incremental version of SDN versus knowing K in advance[9]. Table 5 presents the peak memory consumption of the incremental and plain (i.e., non-incremental) SDN for different K values. The former utilizes only 2% to 6% more space. The dominant factor in space consumption is heap size, which is the same in both methods. The extra space in the incremental version is due to storing nodes and objects that

[9]Note that the two versions have identical I/O cost and practically the same CPU time, i.e., the performance of incremental SDN matches the SDN curves in Fig. 15.

Table 5: Memory usage of Incremental vs. Non-Incr. SDN (KB)

SDN	$K=1$	$K=2$	$K=3$	$K=4$	$K=5$	$K=6$
Incr.	21.3	22.7	24.7	26.4	28.7	30.6
Plain	20.8	21.9	24.1	25.8	27.5	28.9

Table 6: All-DN results

Dataset	CPU time (sec)		Memory (MB)	
	ADN	sADN	ADN	sADN
LB	1	1714	3.6	2.6
TCB	40.5	25389	52.6	25
CA	67.8	49967	82.3	46.9
Synthetic	5.1	10823	9.5	4.8

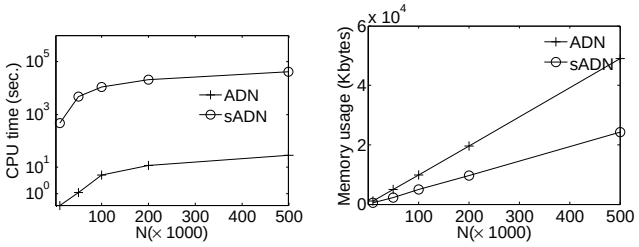
would normally be pruned. Compared to the heap size, this overhead is minimal. When $K = 6$, for example, the maximum number of heap entries is 1165, while the number of pruned nodes/objects is 69.

All-DN results. We examine the performance of our all-DN algorithm described in Sec. 5, which we denote as ADN. For baseline, we construct a competitor, termed *straightforward ADN* (sADN), that invokes SDN for each object in the dataset. For fairness, our implementation of sADN utilizes Observation 2, i.e., DNs are computed only for two of the stripes and two of the quadrants. Here we focus on CPU time as the main performance metric, since both algorithms load the entire dataset in memory. We also measure the space requirements to verify practicality. As shown in Table 6, ADN achieves substantial performance gains over sADN for all datasets, owing to two factors.

First, for every object $r \in S$, sADN constructs four segment trees to store the DNs in the stripes. This is repeated for every object. ADN avoids this deficiency by maintaining two global segment trees. Furthermore, ADN finds the stripe DNs with negligible effort (recall that it identifies as south stripe DNs every pair of successive objects in the leaf node lists in T_x). The west stripe DNs are also computed inexpensively, while inserting r into T_y.

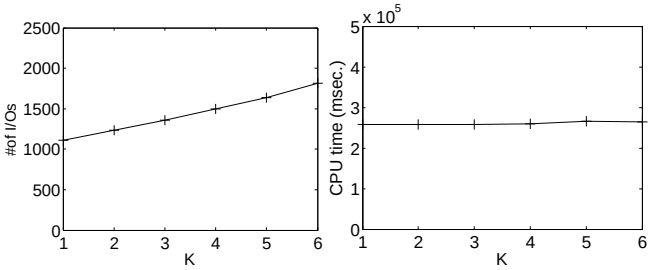
Second, sADN needs to perform dominance checks in two quadrants of every object $r \in S$. In contrast, ADN finds the candidate SW (or NW) DNs of each object r_i with a linear scan of the leaf nodes in T_y that cover the y-extent between r_i and r_{i-1} (r_{i+1}, respectively), i.e., between r_i and the preceding (succeeding) object in the object list L_k^x that includes r_i. Filtering false positives is also performed with a simple comparison per candidate. Obviously, an arithmetic comparison is much cheaper than a dominance check. In LB, for instance, ADN scans 62 leaves in T_y per object on the average, whereas sADN performs 515 dominance checks per object. This leads to another significant gain for ADN.

Turning to the memory consumption in Table 6, we observe that ADN uses about twice the space of sADN. This is because ADN maintains two segment trees in memory. Even so, the space requirements of ADN are well within the capacity of modern PCs. For example, ADN occupies around 82 MBytes for the CA dataset, which contains 2.2M MBRs. As explained in Sec. 5.2, under strict memory constraints, we could resort to a disk-based segment tree [29].



(a) CPU time (b) Memory

Figure 16: All-DN, effect of N (Synthetic dataset)



(a) # of I/Os (b) CPU time

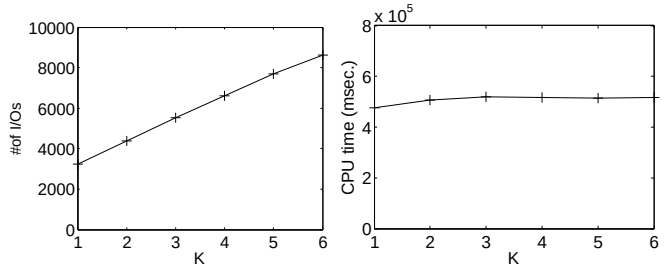
Figure 17: K-DN search in 3-D, effect of K (HOTEL dataset)

Continuing the evaluation of ADN, in Fig. 16 we examine scalability with dataset cardinality N. ADN is consistently three orders of magnitude faster than sADN (between 1300 and 4500 times), with double the memory footprint.

7.3. Experiments in Higher Dimensions

The remaining experiments evaluate our methodology in higher dimensions. We use HOTEL as the four-dimensional dataset, and we also extract its first three dimensions to form the three-dimensional dataset. CNS does not apply here (because NS methods exist only for two dimensions). Hence, we focus on the nature of the problem and on the performance of SDN.

In Fig. 17 and 18 we assess the performance of SDN versus K on three-dimensional and four-dimensional data, respectively. The trend observed here is similar to Fig. 15, but the cost is considerably higher than in two dimensions. The main reason is that the number of DNs increases with d, as elaborated in Sec. 6.1. For instance, the number of DNs ($K = 1$) in the two-dimensional synthetic dataset with 100K objects is 24, versus 673 in three dimensions (for the same dataset cardinality). In the HOTEL dataset, the number is 7308 in three dimensions, and 10385 in four dimensions. Another reason is that, for a fixed cardinality, space becomes sparser in higher dimensions [31], which causes the search area to expand. Finally, the R-tree structure itself degrades with dimensionality, thus reducing the effectiveness of pruning. For a given dataset and R-tree structure, however, SDN is I/O optimal, i.e., the reported I/O



(a) # of I/Os (b) CPU time

Figure 18: K-DN search in 4-D, effect of K (HOTEL dataset)

cost is the smallest possible by any exact, non-precomputation algorithm.

8. Conclusion

In this work, we introduce a new query, called direct neighbor (DN) search. Two objects in a dataset are DNs if it is possible for a window query to overlap these objects and no other. A DN query retrieves all the DNs of a given source object. DN search and its variants, K-DN and all-DN, have wide applicability in competitor analysis. We present algorithms for DN, K-DN and all-DN search. Experiments on real and synthetic data verify that our algorithms vastly outperform baseline solutions built upon existing work.

A direction for future work is DN search for containment queries that are not axis-parallel windows but arbitrary regions. Another challenging direction regards the exploding number of DNs with dimensionality – it would be useful to devise techniques that prioritize among the DNs and possibly choose/compute only a subset of them.

Acknowledgement

This work is funded in the Singapore Management University through research grant 11-C220-SMU-003 from the Ministry of Education Academic Research Fund Tier 1.

References

[1] N. Roussopoulos, S. Kelley, F. Vincent, Nearest neighbor queries, in: SIGMOD, 1995, pp. 71–79.

[2] F. Korn, S. Muthukrishnan, Influence sets based on reverse nearest neighbor queries, in: SIGMOD, 2000, pp. 201–212.

[3] C. S. Fleisher, B. E. Bensoussan, Business and Competitive Analysis: Effective Application of New and Classic Methods, FT Press, 2007.

[4] Apartments - floor preference?, Website, http://www.indianrealestateboard.com/forums/showthread.php/7938-Apartments-Floor-Preference.

[5] A. Guttman, R-trees: A dynamic index structure for spatial searching, in: SIGMOD, 1984, pp. 47–57.

[6] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, The R*-tree: An efficient and robust access method for points and rectangles, in: SIGMOD, 1990, pp. 322–331.

[7] I. Kamel, C. Faloutsos, On packing r-trees, in: CIKM, 1993, pp. 490–499.

[8] S. T. Leutenegger, J. M. Edgington, M. A. Lopez, Str: A simple and efficient algorithm for r-tree packing, in: ICDE, 1997, pp. 497–506.

[9] G. R. Hjaltason, H. Samet, Distance browsing in spatial databases, ACM Trans. Database Syst. 24 (2) (1999) 265–318.

[10] K. C. Lee, W. C. Lee, H. V. Leong, Nearest surrounder queries, in: ICDE, 2006, pp. 85–94.

[11] B. Seo, R. Zimmermann, Edge indexing in a grid for highly dynamic virtual environments, in: ACM Multimedia, 2006, pp. 402–411.

[12] J. Chomicki, Preference formulas in relational queries, ACM Transactions on Database Systems (TODS) 28 (4) (2003) 427–466.

[13] K. Stefanidis, G. Koutrika, E. Pitoura, A survey on representation, composition and application of preferences in database systems, ACM Transactions on Database Systems (TODS) 36 (3) (2011) 19.

[14] G. Koutrika, Y. Ioannidis, Personalizing queries based on networks of composite preferences, ACM Transactions on Database Systems (TODS) 35 (2) (2010) 13.

[15] R. Torlone, P. Ciaccia, Finding the best when it's a matter of preference., in: SEBD, 2002, pp. 347–360.

[16] W. Kießling, Foundations of preferences in database systems, in: VLDB, 2002, pp. 311–322.

[17] W. Kießling, M. Endres, F. Wenzel, The preference SQL system-an overview., IEEE Data Eng. Bull. 34 (2) (2011) 11–18.

[18] S. Börzsönyi, D. Kossmann, K. Stocker, The skyline operator, in: ICDE, 2001, pp. 421–430.

[19] K.-L. Tan, P.-K. Eng, B. C. Ooi, Efficient progressive skyline computation, in: VLDB, 2001, pp. 301–310.

[20] D. Papadias, Y. Tao, G. Fu, B. Seeger, Progressive skyline computation in database systems, ACM Trans. Database Syst. 30 (1) (2005) 41–82.

[21] M. Sharifzadeh, C. Shahabi, L. Kazemi, Processing spatial skyline queries in both vector spaces and spatial network databases, ACM Transactions on Database Systems (TODS) 34 (3) (2009) 14.

[22] K. Deng, X. Zhou, H. T. Shen, Multi-source skyline query processing in road networks, in: ICDE, 2007, pp. 796–805.

[23] L. Chen, X. Lian, Efficient processing of metric skyline queries, IEEE Trans. Knowl. Data Eng. 21 (3) (2009) 351–365.

[24] D. Papadias, Y. Tao, Reverse nearest neighbor query, in: Encyclopedia of Database Systems, 2009, pp. 2434–2438.

[25] A. Vlachou, C. Doulkeridis, Y. Kotidis, K. Nørvåg, Reverse top-k queries, in: ICDE, 2010, pp. 365–376.

[26] E. Dellis, B. Seeger, Efficient computation of reverse skyline queries, in: VLDB, 2007, pp. 291–302.

[27] M. de Berg, O. Cheong, M. van Kreveld, M. Overmars, Computational Geometry: Algorithm and Applications, Springer-Verlag, 2008.

[28] J. L. Bentley, D. Wood, An optimal worse case algorithm for reporting intersections of rectangles, IEEE Trans. on Computers C-29 (7) (1980) 571–577.

[29] L. Arge, D. E. Vengroff, J. S. Vitter, External-memory algorithms for processing line segments in geographic information systems, Algorithmica 47 (1) (2007) 1–25.

[30] Z. Zhang, Y. Yang, R. Cai, D. Papadias, A. K. H. Tung, Kernel-based skyline cardinality estimation, in: SIGMOD, 2009, pp. 509–522.

[31] K. S. Beyer, J. Goldstein, R. Ramakrishnan, U. Shaft, When is 'nearest neighbor' meaningful?, in: ICDT, 1999, pp. 217–235.

[32] B. R. Vatti, A generic solution to polygon clipping, Commun. ACM 35 (7) (1992) 56–63.

[33] M. Levandowsky, D. Winter, Distance between sets, Nature 234 (1971) 34–35.

Appendix

This appendix provides pseudo-code for the complete DN algorithm described in Sec. 3.3. First, in Algorithm 3 we sketch its visibility check building block, used for stripe DNs. Note that in Line 3, leaf n_+ is the one that covers value $r.y_h$ (for east/west stripe) or $r.x_h$ (for north/south stripe).

The complete DN algorithm, using SDN quadrant search, is given in Algorithm 4. We traverse the R-tree on S from the root, and insert each encountered R-tree entry e into a min-heap

Algorithm 3: IsStripeDN(st,r,T)

Input: a stripe st of q, an object (or an R-tree node's MBR) r, segment tree T corresponding to st
Output: a boolean value indicating whether r is a DN

1 issue a stabbing queries on T at (1) $r.y_l$ for the east and west stripes, or (2) $r.x_l$ for the north and south stripes;
2 let n_- be the leaf containing the query point;
3 **for** *each leaf* n_i *between* n_- *and* n_+ **do**
4 let L_i be the object list for n_i;
5 **if** $|L_i| = 0$ **then** return $true$;
6 return $false$;

sorted on *mindist(q,e)*, i.e., the minimum distance between e and q. The top entry e is popped from the heap iteratively, and processed according to whether e intersects q in any of the stripes and quadrants. If e corresponds to an internal R-tree node that is not pruned in lines 27-28 or 38-39, we fetch this node from the disk and push its child entries into the heap; if e is an un-pruned data object, we output it as a DN. This process is repeated until the heap is empty.

Algorithm 4: DN search

Input: an R-tree on S, a source object q
Output: set $\mathcal{R}_{\mathcal{DN}}$ of the DNs of q

1 initialize empty result sets R_E,R_W,R_N,R_S for the east, west, north, south stripes, and R_{NE},R_{NW},R_{SW}, R_{SE} for the northeast, northwest, southwest, southeast quadrants of q; and an empty result set $\mathcal{R}_{\mathcal{DN}}$;
2 initialize four empty segment trees T_E, T_W, T_N, T_S;
3 initialize an empty min-heap H;
4 insert all the entries in the R-tree root into H;
5 **while** H *is not empty* **do**
6 remove the top entry e from H;
7 **if** e *is completely inside* q **then**
8 if e is a data entry then insert it into $\mathcal{R}_{\mathcal{DN}}$;
9 else, read the whole sub-tree rooted at e and insert all of its data objects into $\mathcal{R}_{\mathcal{DN}}$;
10 **if** e *partially overlaps* q **then**
11 **if** e *is a data entry* **then**
12 if e intersects X, where $X \in \{E, W, N, S, NE, NW, SW, SE\}$, insert e into R_X;
13 **else**
14 fetch the node n pointed by e;
15 insert all the entries in n into H;
16 **if** e *is completely outside* q **then**
17 **if** e *intersects one of the 4 stripes* **then**
18 let $X \in \{E, W, N, S\}$ be that stripe;
19 **if** *IsStripeDN(*X,e,T_X*)=*$false$ **then**
20 discard e;
21 **else**
22 **if** e *is a data entry* **then**
23 insert e into R_X;
24 insert e into T_X;
25 **else**
26 fetch the node n pointed by e;
27 insert all the entries in n into H;
28 **if** e *intersects one of the 4 quadrants* **then**
29 let $X \in \{NE, NW, SW, SE\}$ be that quad.;
30 **if** e *is dominated by an element in* R_X **then**
31 discard e;
32 **else**
33 **if** e *is a data entry* **then**
34 insert e into R_X;
35 **else**
36 fetch the node n pointed by e;
37 **for** *each entry* e' *in* n **do**
38 **if** e' *is dominated by an element in* R_X **then**
39 discard e';
40 **else**
41 insert e' into H;
42 return $\mathcal{R}_{\mathcal{DN}} = \mathcal{R}_{\mathcal{DN}} \cup \bigcup_{X \in \{E,W,N,S,NE,NW,SW,SE\}} R_X$;